CS 5830 Homework 2

Instructor: Thomas Ristenpart
TAs: Andres Fabrega, Sanketh Menda, and Dhrumil Patel
Due: 7 October 2022

Submission and Grading

Submit your written answers to Gradescope and your code answers to CMSX. You must submit appropriate answers to the both submission sites for full credit.

Problems in this assignment are of two types:

1. *Attacking cryptography.* Given an incomplete template file, you need to fill in the missing pieces to mount an attack, e.g. retrieving a flag or forging a ciphertext.
2. *Talking cryptography.* Talk us through how you attacked a given cryptographic primitive or how you would hypothetically attack it.

The exact materials and the format you would need to submit them in for a given problem are indicated at the end of the problem.

Late Days

Homeworks are due on the due date by 11:59pm EST. You can use in total 3 late days throughout the semester.

Academic Integrity

Feel free to refer to external materials, but as usual please acknowledge them in your writeup in accordance with the Cornell Code of Academic Integrity.

Server Availability

To query the server, you'll use the same username and API key throughout this semester. Your username is your Cornell NetID (like `sm2289`) and you can find your assigned API key on CMSX, in the *Grading Comments & Requests* section of the placeholder assignment called `API_keys`.

We made a best effort to ensure the reliability of the challenge servers, but unfortunately, due to the nature of software and the internet, there may be disruptions. If you run into a disruption (the server is not responding, it is providing an error code, etc.), please do not panic, and reach out to the TAs (`@sanketh`, `@Andres`, or `@Dhrumil Patel`) on Slack. If this happens within 24 hours of the deadline, we will extend the deadline to accommodate for the inconvenience.

Verbosity for Written Answers

Generally, we prefer concise answers.

For a question like "Explain how you mount the attack?" we are expecting a high-level, but precise description of the attack in 3-10 sentences. Here is an example of a good answer.

> I started with an initial guess of the 16-byte all-zeros message. Then for each of the 16 bytes, I did X. Then did Y. Since the success of this is not guranteed, I repeated the process 20 times and aggregated. With the confidence, I did Z to recover the `FLAG`.

For a question like "Estimate the number of queries your attack issues?" we want you to think about how many queries you *really* need, and expect a high-level answer with ballpark numbers. Here is an example of a good answer.

> My attack does 5 queries initially for [purpose]. Then it does 1 queries per `FLAG` byte. In sum, for a flag of length N, it needs 1 + 5/N queries per `FLAG` byte.

Flag Notes

Each problem includes a description of the expected flag. For instance, it may say that it has 10 alphanumeric characters. Feel free to use that to restrict your search space, and to verify that you found the right flag (if you find something that meets the flag description, it is overwhelmingly likely that you found the right flag).

During grading, we will run your code against a different flag that nevertheless meets the description. So please make sure that you are not hardcoding assumptions regarding your specific flag.

Question 1. Malleability Against SHA256 into CTR

Introduction. Alice runs a short-form social video service *add-rotate-xor*. When a user logs into the service, they are given an encrypted cookie with their username and a validity of one hour, it has the form

```
username=alice&validtill=1645220443
```

The cookie is encrypted with CTR mode[1] and checked before a video can be watched.

Alice decides to add a premium ad-free option. Rather than updating the entire backend to support this new feature, she decides to just add a new boolean `adfree` field to the login cookie and check it before streaming the video. The new cookies look like

```
username=alice&validtill=1645220443&adfree=0
```

Before she deploys the system, she realizes that CTR mode is malleable, so people can tamper with the encrypted cookie, even if they don't know the secret key. So, at the last moment, she decides to tack on a SHA256 hash to the cookie. The hash is added to the cookie when generated and verified before reading the cookie. The final cookie looks like

```
username=alice&validtill=1645220443&adfree=0&sha256=3cd0...86d5
```

where

```
SHA256(b"username=alice&validtill=1645220443&adfree=0") =
    3cd0e601f8e7e3660049c7e5fdc4fea6bee54343b465181078748be8081686d5
```

SHA256 is a cryptographic hash, so it should be impossible to tamper with the encrypted cookie, right? Wrong, SHA256 is still a public function, anyone who knows the plaintext can generate a valid SHA256 value for it.

Problem Oracle. We are given two oracles,

1. `genCookie` that generates a fresh encrypted cookie for you, with your `NetID` as `username`, a 10-digit unix timestamp and `adfree=0`.
2. `isCookieAdFree` that takes an encrypted cookie, verifies it (does it decrypt with CTR, is the sha256 hash valid, and is the format correct), and returns `True` if the ciphertext is valid and `adfree=1`, and `False` otherwise.

The Goal. We want to take the encrypted `adfree=0` cookie and tamper with it to generate a valid `adfree=1` cookie. We know what the plaintext of the cookie looks like, we know the `username`, we know the `adfree` value, and can guess the `validtill` valid with reasonable accuracy: it is an hour from now plus or minus say 60 seconds. We can use this information to generate modified ciphertexts and one of them is likely to pass validation.

[1] Recall that, in CTR mode, the first block of the ciphertext corresponds to the nonce.

Query Estimates. This attack should ideally take one `genCookie` query and a hundred or so `isCookieAdFree` queries.

Flag Format. There is no flag for this question.

What to Submit.

- (5 pts) *Source code.* Copy the file `hw2_q1_ctr_sha_forgery.py` into your homeworks' folder and complete the TODOs to conduct the attack. Submit `hw2_q1_ctr_sha_forgery.py` to CMSX.
- (5 pts) *Written answer.* Submit your answers to the following questions to Gradescrope:
    1. (3 pts) Explain how you mount the attack?
    2. (1 pts) Estimate the number of queries your attack issues?
    3. (1 pts) Explain why IND-CPA security does not suffice for preventing your attack. How would you modify Alice's scheme to prevent this attack?

Warning. We will be running your code with a different username when grading, so use `cfg.usr` rather than hardcoding your NetID into the solution.

Question 2. Malleability Against Truncated HMAC-SHA256 into CTR

Introduction. Shortly after deploying the system from Question 1, Alice gets a bug bounty entry from a helpful security researcher informing her that the current cookie scheme can be broken. She sighs, awards the researcher a bug bounty for bringing it to her attention, and starts thinking about how to resolve the issue. After hours of googling and talking with friends, she realizes that she should be using a keyed-HMAC instead of plain SHA256. So, she updates her backend infrastructure to provision a new key and updates it so the cookie now includes a HMAC hash. The upgraded cookies look like

```
username=alice&validtill=0000000000&adfree=0&hmacsha256=3939...884f
```

where

```
HMAC-SHA256(
    key=b"yellow submarine",
    value=b"username=alice&validtill=0000000000&adfree=0"
    ) = 39395b8f486425d52d87813c3e51c39d895cd2c236652c08d67b7da8d8ed884f
```

But, these hash values are too long, they barely fit on a line, and she noticed sites like GitHub use truncated hash values, so, as part of the upgrade, she decides to truncate the hash values to the first 2-characters. The upgraded cookies look like

```
username=alice&validtill=0000000000&adfree=0&hmacsha256=39
```

Neat, she hits deploy. HMAC-SHA256 is secure, so this should be secure right? Unfortunately not, even if HMAC-SHA256 is secure, the truncation makes the HMAC value easy to bruteforce.

Note. To make it easier on our servers, for this question, we have fixed the `validtill` value to be `0000000000` (and you can hardcode this assumption into your code.)

Problem Oracle. We are given two oracles,

1. `genCookie` that generates an encrypted cookie for your, with your `NetID` as `username`, `validtill=0000000000` and `adfree=0`.
2. `isCookieAdFree` that takes an encrypted cookie, verifies it (does it decrypt with CTR, is the HMAC-SHA256 valid, and is the format correct), and returns `True` if the ciphertext is valid and `adfree=1`, and `False` otherwise.

The Goal. As before, we want to take the encrypted `adfree=0` cookie and tamper with it to generate a valid `adfree=1` cookie.

Query Estimates. This attack should ideally take one `genCookie` query and about a thousand `isCookieAdFree` queries.

Hint. If your attacks seems a lot more expensive (say 100x) than the expected thousand queries, consider making sure that you are using the fact that the hmac value is hex encoded, so each character falls in a narrower range than an arbitrary byte does. On a piece of paper, write down random hex values and observe their xors.

Flag Format. There is no flag for this question.

What to Submit.

- (5 pts) *Source code.* Copy the file `hw2_q2_ctr_hmac_forgery.py` into your homeworks' folder and complete the TODOs to conduct the attack. Submit `hw2_q2_ctr_hmac_forgery.py` to CMSX.
- (5 pts) *Written answer.* Submit your answers to the following questions to Gradescrope:
    1. (2 pts) Explain how you mount the attack?
    2. (1 pts) Estimate the number of queries your attack issues?
    3. (2 pts) HMAC is conjectured to be a secure in the sense of being a pseudorandom function (PRF), but your attack succeeds at forging against truncated HMAC. Explain why this is not a contradiction and what are the implications on design of message authentication schemes.

Warning. We will be running your code with a different username when grading, so use `cfg.usr` rather than hardcoding your NetID into the solution.

Question 3. CBC Padding Oracle Attack

This is, by far, one of the most recognizable attacks in modern symmetric cryptography. To illustrate how awesome the attack is: OpenSSL (one of the most widely-used cryptographic libraries) originally patched it in 2003,[2] then had to re-patch it in 2013,[3] and again in 2014,[4] and again in 2016.[5]

Introduction. See lecture 7 (slides).

Problem Oracle. We are given two oracles,

1. `encrypt` produces an encryption of the `FLAG` with a fresh IV,

```
iv = GenerateRandomIV()
ciphertext = AES128-CBC-Encrypt(key, iv, PKCS7Padding(FLAG))
return iv || ciphertext
```

2. `checkPadding` that takes `iv || ciphertext` and works as follows

```
message = AES128-CBC-Decrypt(key, iv, ciphertext)
if ValidPKCS7Padding(message):
    return true
return false
```

The Goal. Use the padding oracle (`checkPadding`) to extract `FLAG`. In addition to lecture 7, you may find Section 3 of Vaudenay (2002) helpful. This is the first attack in this course that requires non-trivial coding, so I highly recommend planning your code on paper (like what functions you want to implement), document them, write tests, and then do the implementation.

Query Estimates. This attack should ideally take a few thousand queries.

Flag Format. `Pad{[2 random bytes in hex]}`, like `Pad{ffff}`.

What to Submit.

- (2 pts) *Source code.* Copy the file `hw2_q3_padding_oracle.py` into your homeworks' folder and complete the TODOs to conduct the attack. Submit `hw2_q3_padding_oracle.py` to CMSX.
- (3 pts) *Captured flag.* Put the captured `FLAG`, just the `FLAG` no quotes or special characters in text (not hex) in a new text file `hw2_flag_padding.txt` and submit it to CMSX.
- (5 pts) *Written answer.* Submit your answers to the following questions to Gradescope:
    1. (2 pts) Explain how you found the flag, including an estimate of the number of queries your attack uses.

[2] CVE-2003-0078
[3] "Lucky Thirteen" CVE-2013-0169
[4] "POODLE" CVE-2014-3566
[5] CVE-2016-2107

2. (3 pts) Consider using a MAC-then-Encrypt approach to mitigate CBC padding oracle vulnerabilities. Specifically, a colleague suggests transitioning to the following symmetric encryption scheme:

Encrypt($(K_a K_e)$, M):
$IV \xleftarrow{\$} \{0,1\}^n$
$T \leftarrow \text{HMAC}(K_a, M)$
$X_{pad} \leftarrow \text{PKCS7Padding}(M \| T)$
$C \leftarrow \text{AES128-CBC-Encrypt}(K_e, IV, X_{pad})$
Return (IV, C)

Decrypt($(K_a K_e)$, (IV, C)):
$X_{pad} \leftarrow \text{AES128-CBC-Decrypt}(K_e, IV, C)$
If ValidPKCS7Padding(X_{pad}) then
    $X \leftarrow \text{RemovePCKS7Padding}(X_{pad})$
    Parse X as M and T, where $|T| = 64$ bytes
    $T' \leftarrow \text{HMAC}(K_a, M)$
    If $T = T'$ then Return M
Return $\perp$

Here K_e is a AES-CBC encryption key and K_a is a secret key for use with HMAC. Would this approach prevent padding oracle attacks? If not, explain how an attack could recover plaintext data. If yes, explain why it prevents padding oracle attacks.

Evaluation

We would love it if you could provide us feedback to improve future homeworks. Please provide answers to the following questions on Gradescrope.

- Did you find the homework easy, appropriately difficult, or too difficult?
- How many hours did you spent on this homework?
- Did you feel that the coding was too much, appropriate, or not enough?

And feel free to include additional feedback (on the homework or logistics in general). Also, this is not a tight deadline, we'd be delighted to hear your feedback via Slack before or after the deadline.